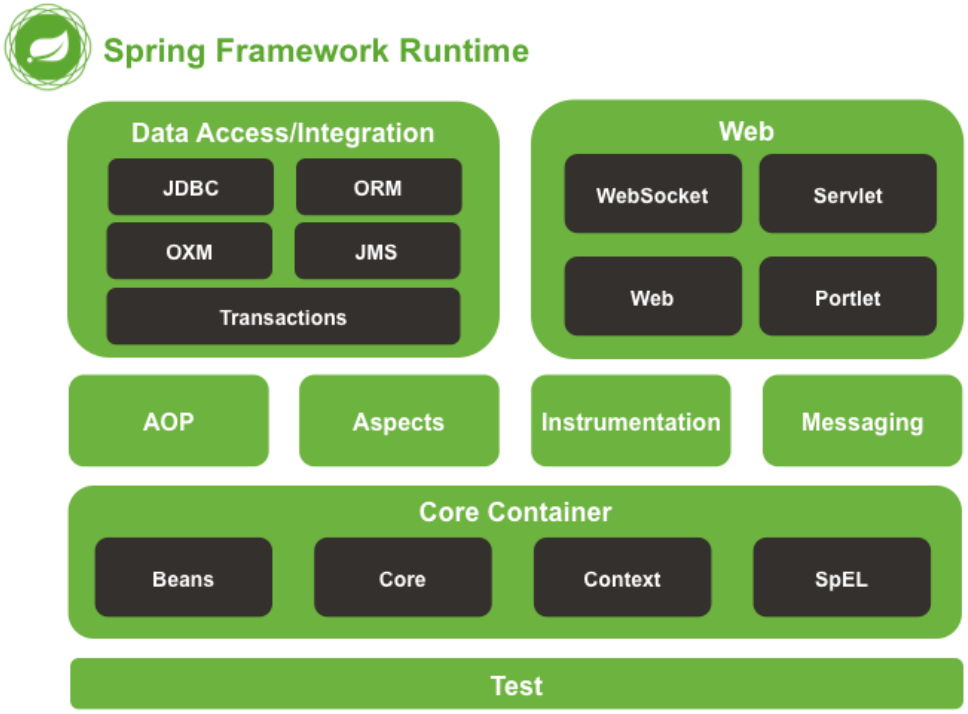
SPRING FRAMEWORKS



Capitulo 4
Spring MVC

Contenido

1 ARQUITECTURA Y CONFIGURACIÓN 4

1.1 ARQUITECTURA 4
1.2 CONFIGURACIÓN 5
1.2.1 Configuración del servlet 5
1.2.2 Configuración del spring-context 6

2 CONTROLADORES 9

2.1 MODEL Y VIEW 9
2.2 ANOTACIONES 9
2.3 @CONTROLLER 10
2.4 @REQUESTMAPPING 10
2.5 @PATHVARIABLE 11
2.6 @REQUESTPARAM 12
2.7 @REQUESTHEADER 13
2.8 @COOKIEVALUE 13

3 EJERCICIO 1 14

4 MAPEO DE PETICIONES 15

4.1 ESTABLECER RUTA BASE 15
4.2 MÉTODO DE LA PETICIÓN 15
4.3 PARÁMETROS EN LA PETICIÓN 16
4.4 HEADERS DE LA PETICIÓN 17

5 EJERCICIO 2 18

6 LAS VISTAS 19

6.1 VISTAS 19
6.2 ERRORES Y EXCEPCIONES 20
6.3 REDIRECCIÓN 21
6.4 TAG LIBRARY 21
6.5 SPRING:MESSAGE 22
6.6 FORM:FORM 22
6.7 CONTROLES 23

7 EJERCICIO 3 25

8 CAPA DE SERVICIOS 25

8.1 EL APPLICATION CONTEXT 25
8.2 CONFIGURACIÓN 26
8.3 EJEMPLO ILUSTRATIVO 27

9 EJERCICIO 4 30

10 OTROS ASPECTOS 30

10.1 @REQUESTBODY 30
10.2 @RESPONSEBODY 31
10.3 @MODELATTRIBUTE 31
10.4 HANDLERS 33
10.4.1 Tipos de Argumentos 33
10.4.2 Tipos Retornados 34
10.4.3 Ejemplos 35
10.5 MODELOS 36
10.5.1 Model 36
10.5.2 @SessionAttributes 37
10.5.3 @ModelAttribute 37
10.5.4 @InitBinder 38

1 ARQUITECTURA Y CONFIGURACIÓN

1.1 Arquitectura

En la Figura 1 se tiene la arquitectura del funcionamiento de una aplicación con Spring MVC.

Se puede identificar claramente que en Spring MVC, el servlet **DispatcherServlet** funciona bajo el patrón **front controller**. El patrón front controller proporciona un punto de entrada único. De manera que todas los request son procesados por un mismo Servlet, en el caso de Spring MVC, se trata de DispatcherServlet. Este servlet se va a encargar de gestionar toda la lógica en la aplicación.

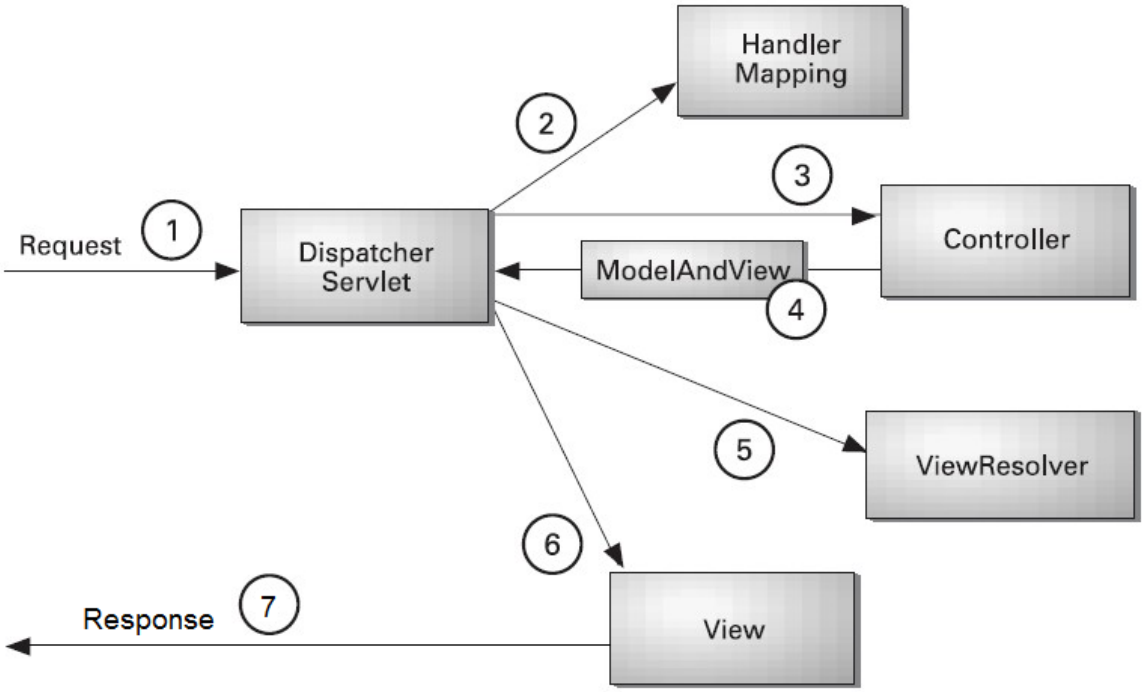


Figura 1

El flujo básico en una aplicación bajo Spring MVC es el siguiente:

1. El request llega al **DispatcherServlet** (1)
2. El DispatcherServlet tendrá que encontrar el controlador va a tratar la petición. Para ello el DispatcherServlet tiene que encontrar el manejador asociado a la url de la petición. Todo ésto se realiza en la fase de **HandlerMapping** (2).
3. Una vez encontrado el Controller, el DispatcherServlet le dejará gestionar a éste la petición (3). En el controlador se deberá realizar todo la lógica de negocio de nuestra

aplicación, es decir, aquí se llamara a la capa de servicios. El controlador devolverá al Dispatcher un objeto de tipo **ModelAndView**. El **Model** representa los valores que que se obtienen de la capa de servicio y **View** será el nombre de la vista en la que se debe mostrar la información que va contenida dentro de ese Model.

4. Una vez pasado el objeto **ModelAndView** al **DispatcherServlet**, será éste el que tendrá que asociar el nombre de la vista retornada por el controlador a una vista concreta, en este caso una página jsp. Este proceso es resuelto por el ViewResolver (4).
5. Finalmente y una vez resuelta la vista, el DispatcherServlet tendrá que pasar los valores del **Model** a la vista concreta, en este caso la pagina jsp, View (5).

1.2 Configuración

1.2.1 Configuración del servlet

El dispacher es un servlet que se debe configurar en el descriptor de despliegue (web.xml), un ejemplo es el que se muestra en el Script 1.

Script 1

```
<web-app . . . >

  <servlet>
    <servlet-name>spring</servlet-name>
    <servlet-class>
       org.springframework.web.servlet.DispatcherServlet
    </servlet-class>
    <load-on-startup>1</load-on-startup>
  </servlet>

  <servlet-mapping>
    <servlet-name>spring</servlet-name>
    <url-pattern>*.html</url-pattern>
  </servlet-mapping>

</web-app>
```

Es necesario configurar el spring-context para el DispacherServlet, en este caso se debe ubicar en la misma carpeta del archivo web.xml (WEB-INF) y debe tener el nombre spring-servlet.xml.

El mapeo indica que solo responde a request que finalicen con **".html"**. Otra opción para configurar el dispacher se muestra en el Script 2.

Para este segundo caso, el archivo de contexto puede tener cualquier nombre, ya que es un parámetro del DispacherServlet.

El mapeo indica que procesa todos los request.

Script 2

```
<web-app . . . >

  <servlet>
    <servlet-name>appServlet</servlet-name>
    <servlet-class>
      org.springframework.web.servlet.DispatcherServlet
    </servlet-class>
    <init-param>
      <param-name>contextConfigLocation</param-name>
      <param-value>
        /WEB-INF/spring/appServlet/servlet-context.xml
      </param-value>
    </init-param>
    <load-on-startup>1</load-on-startup>
  </servlet>

  <servlet-mapping>
    <servlet-name>appServlet</servlet-name>
    <url-pattern>/</url-pattern>
  </servlet-mapping>

</web-app>
```

1.2.2 Configuración del spring-context

En el archivo spring-context se debe configurar como minimo lo siguiente:

- Configurar el ViewResolver.
- Habilitar el uso de anotaciones.
- Configurar la carpeta de clases contraladoras.

Configuración del ViewResolver

Script 3

```
<beans:bean
class="org.springframework.web.servlet.view.InternalResourceViewResolver">
  <beans:property name="prefix" value="/WEB-INF/views/" />
  <beans:property name="suffix" value=".jsp" />
</beans:bean>
```

Cuando en el objeto ModelAndView indicamos el nombre de un view, solo especificamos el nombre del archivo, su extensión y ubicación se configura en el ViewResolver, un ejemplo se tiene en el Script 3.

El Script 3 esta configurando que los views, las páginas jsp se deben guardar en la carpeta /WEB-INF/views.

Habilitar el uso de anotaciones

Para poder usar la anotación @Controller en las clases controladoras se debe habilitar su uso en el spring-context, tal como se ilustra en el Script 4.

Script 4

```
  <annotation-driven />
```

Configurar la carpeta de clases contraladoras

Se debe también configurar el paqueque donde spring buscará las clases controladoras, tal como se ilustra en el Script 5.

Script 5

```
  <context:component-scan base-package="pe.egcc.demomvc.controller" />
```

Clase controladora

En el Script 6 se tiene un ejemplo de una clase controladora.

En el objeto de tipo Model se tienen los datos que se enviaran al view, en este caso se envía un saludo.

El método home() retorna el nombre del view, en este caso "home", esto quiere decir que en la carpeta "/WEB-INF/views" debe existir el archivo home.jsp.

Script 6

```
package pe.egcc.demomvc.controller;

import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;

@Controller
public class HomeController {

   @RequestMapping(value = "/", method = RequestMethod.GET)
   public String home( Model model ) {
      model.addAttribute("mensaje", "Hola GUSTAVO CORONEL." );
      return "home";
   }

}
```

Los view son generalmente archivos jsp, para el caso del Script 6, sería por ejemplo un archivo jsp de nombre home.jsp, el Script 7 muestra un ejemplo de lo que podría ser la codificación de esta vista.

Script 7

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<%@ page session="false" %>
<html>
<head>
   <title>Home</title>
</head>
<body>
   <h1>SALUDO</h1>
   <p>${mensaje}</p>
</body>
</html>
```

2 CONTROLADORES

Los controladores son las clases que reciben las peticiones de los usuarios, llaman al servicio que implementa la lógica de negocio, y deciden la vista a aplicar para mostrar la información resultante. Son equivalentes a los clásicos servlets, pero que gracias a Spring pueden escribirse de una forma mucho más intuituiva.

2.1 Model y View

Las clases Model y View son los elementos básicos con los que trabaja un controlador. El modelo almacena los datos del proceso y la vista decide su representación.

Un modelo en Spring es simplemente un array asociativo. Es decir, una colección de pares <nombre,valor>. Lo que en Java se corresponde con el tipo Map. Esto permite una gran flexibilidad, ya que dicho modelo se puede convertir fácilmente a cualquier otra clase según la tecnología que se quiera utilizar, como por ejemplo al formato de atributos que espera una página JSP.

La vista en Spring es una cadena de texto con un nombre. El mecanismo de resolución de dicho nombre es totalmente configurable, y la salida puede ser el resultado de aplicar una plantilla JSP, o una salida personalizada utilizando XML, JSON, o cualquier otro tipo de formato.

2.2 Anotaciones

Spring ofrece una forma muy conveniente de utilizar anotaciones para definir controladores. Pero es necesario recordar activar su procesamiento en la configuración, tal como se ilustra en el Script 8.

Script 8

```
<?xml version="1.0" encoding="UTF-8"?>
<beans . . . >

   <mvc:annotation-driven />
   . . .

</beans>
```

2.3 @Controller

Script 9

```
import org.springframework.stereotype.Controller;

@Controller
public class FarmaciaController {
   . . .
}
```

Esta anotación se utiliza para indicar que una clase actúa como controlador, en el Script 9 se ilustra su uso.

Una clase de este tipo no tiene que heredar de ninguna clase, ni implementar ninguna interface específica.

2.4 @RequestMapping

Esta anotación se utiliza para configurar la URL a la que tiene que atender una clase o un método. Si se aplica a una clase, entonces las URLs de sus métodos son relativas a la indicada en la clase. Un método que tenga esta anotación no tiene que seguir ningún patrón específico, un ejemplo ilustrativo se tiene en el Script 10.

Script 10

```
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;

@Controller
@RequestMapping("/recetas")
public class RecetasController {

   @RequestMapping(method = RequestMethod.GET)
   public void handler() {
      . . .
   }

   @RequestMapping(value="/nueva", method = RequestMethod.GET)
   public void nuevaHandler() {
      . . .
   }

}
```

Como se observa en el Script 10, la anotación permite utilizar algunos parámetros adicionales, como el método HTTP concreto. Sólo si la petición HTTP es del tipo indicado se llamará al método.

Otros parámetros de la anotación permiten indicar el formato aceptado según el contenido de la cabecera Content-Type (consumes="application/json"), el formato generado según la cebecera Accept (produces="text/plain"), los parámetros presentes en la URL (params="mode=online"), o la cabecera HTTP (headers="cabecera=personalizada").

Una característica interesante es que la expresión de los atributos también se pueden negar para excluir condiciones en vez de incluirlas (consumes="!text/plain").

2.5 @PathVariable

Las URLs que se configuran para los controladores son en realidad URI Templates. Esta nomenclatura permite definir variables que pueden ser extraídas e inyectadas directamente en los parámetros de los métodos.

Por ejemplo, las URLs de tipo "farmacia/cliente/cantinflas" pueden definirse con una URI Template de la forma "farmacia/cliente/{clienteId}". Y asociar la variable a un parámetro de entrada utilizando la anotación @PathVariable, tal como se ilustra en el Script 11.

Script 11

```
@RequestMapping(value="/cliente/{clienteId}")
public void handler(@PathVariable("clienteId") Long clienteId) {
   . . .
}
```

También se puede usar expresiones regulares para validar el dato, tal como se ilustra en el Script 12, donde el valor de **id** debe ser un número entero, en contrario se tendrá un error HTTP 404.

Script 12

```
@RequestMapping(value="/codigo/{regexp1:[0-9]+}")
public String  codigo(@PathVariable("regexp1") int id, ModelMap model) {
   . . .
}
```

También es posible especificarse varias variables, e incluso combinar las de los métodos con los de la clase, tal como se ilustra en el Script 13.

Para este caso la petición debe ser de la siguiente manera:

```
/cliente/456/receta/38
```

La variable clienteId toma el valor 456 y la variable recetaId toma el valor 38.

Script 13

```
@Controller
@RequestMapping(value="/cliente/{clienteId}")
public class RecetasController {

   @RequestMapping(value="/receta/{recetaId}")
   public void handler(
      @PathVariable("clienteId") Long clienteId,
      @PathVariable("recetaId") Long recetaId) {
      . . .
   }

}
```

2.6 @RequestParam

Esta anotación permite acceder a los parámetros de una petición HTTP, como se ilustra en el Script 14.

Script 14

```
   @RequestMapping(value="/consulta")
   public String handler(@RequestParam("recetaId") Long recetaId) {
      ...
   }
```

Para una petición de la forma:

```
/consulta?recetaId=123
```

El argumento `recetaId` del método tomaría el valor **123** directamente del parámetro de la url. Por otra parte, si el parámetro no fuera obligatorio, se podría indicar con `required=false`.

El Script 15 ilustra el uso de required, en este caso el argumento `recetaId` del método toma valor `null`, por lo que no puede ser de tipo primitivo.

Es necesario tener en cuenta que durante el proceso se hace necesario verificar si el argumento `recetaId` ha tomado valor `null` mediante un estructura `if`.

Script 15

```
  @RequestMapping(value="/consulta")
  public String handler(@RequestParam(value="recetaId", required=false)
Long recetaId) {
     ...
  }
```

2.7 @RequestHeader

Esta anotación permite acceder al valor de una cabecera HTTP, tal como se ilustra en el Script 16.

Script 16

```
  @RequestMapping(value="/cabeza")
  public void handler(@RequestHeader("Accept-Language") String cabecera) {
     ...
  }
```

2.8 @CookieValue

Esta anotación permite acceder al valor de una cookie, tal como se ilustra en el Script 17.

Script 17

```
  @RequestMapping(value="/galleta")
  public void handler(@CookieValue("JSESSIONID") String cookie) {
     ...
  }
```

3 EJERCICIO 1

Desarrollar un proyecto utilizando Spring Framework que permita:

- Calcular el importe de una venta.
- Calcular el promedio de 4 notas.

4 MAPEO DE PETICIONES

En la sección anterior hemos visto la forma básica para generar controladores basados en anotaciones, en esta sección se revisa las opciones para realizar el mapeo de peticiones hacia los métodos de los controladores.

4.1 Establecer ruta base

Se tiene la posibilidad de establecer la ruta base de las peticiones que atenderá un controlador al combinar el uso de la anotación `RequestMapping` a nivel clase y a nivel de método, tal como se ilustra en el Script 18.

Script 18

```
@Controller("userController")
@RequestMapping("/admin/user/*")
public class UserController {

   @RequestMapping(value = "list.htm")
   public ModelAndView list() {
      . . .
   }

   . . .

}
```

El Script 18 indica que el método `list()` de la clase `UserController` atenderá las peticiones realizadas a la URL `/admin/user/list.htm`.

4.2 Método de la petición

También se puede filtrar el mapeo de las peticiones a través del método de la petición, tal como se ilustra en el Script 19.

En el ejemplo del Script 19, la misma URL `/admin/user/filteredList.htm` es mapeada hacia 2 métodos del controlador, `filterForm()` atiende las peticiones de tipo GET (por ejemplo un enlace para mostrar el formulario de filtrado) y `filteredList()` atiende las peticiones de tipo POST (para enviar el formulario con los filtros y mostrar la lista).

Script 19

```
@RequestMapping("/admin/user/*")
public class UserController {

   @RequestMapping(value = "filteredList.htm", method = RequestMethod.GET)
   public ModelAndView filterForm() {
      . . .
   }

   @RequestMapping(value = "filteredList.htm", method = RequestMethod.POST)
   public ModelAndView filteredList() {
      . . .
   }

   . . .

}
```

4.3 Parámetros en la petición

Script 20

```
@Controller("userController")
@RequestMapping("/admin/user/*")
public class UserController {

   @RequestMapping( value="update.htm",
                    method=RequestMethod.GET, params={"create"})
   public ModelAndView create() {
      . . .
   }

   @RequestMapping( value="update.htm",
                    method=RequestMethod.GET, params={"!create"})
   public ModelAndView update(@RequestParam("login") String login) {
      . . .
   }

   . . .

}
```

Otra de las opciones es realizar el mapeo de acuerdo a los parámetros recibidos (o no) dentro de la petición, tal como se ilustra en el Script 20.

Los métodos `create()` y `update()` son mapeados a las peticiones de `/admin/user/update.htm` de tipo `GET`. Sin embargo el método `create()` atenderá

aquellas peticiones en las que el parámetro de petición `create` esté presente y el método `update()` atenderá las peticiones cuando el mismo parámetro esté ausente.

4.4 Headers de la petición

Finalmente, podemos mapear las peticiones de acuerdo a las cabeceras de la petición, tal como se ilustra en el Script 21.

Script 21

```
@Controller("userController")
@RequestMapping("/admin/user/*")
public class UserController {

   @RequestMapping(value = "home.htm", header = "content-type=text/*")
   public ModelAndView home() {
      . . .
   }

   . . .

}
```

También se puede usar el símbolo de admiración (!) para la negación.

5 EJERCICIO 2

Desarrollar un proyecto utilizando Spring Framework que permite calcular el MCD y MCM de dos números enteros.

6 LAS VISTAS

La capa de presentación web se compone de vistas, donde cada vista se identifica por un nombre virtual que Spring resuelve según la tecnología usada por la aplicación, como JSP por ejemplo.

6.1 Vistas

Como ya se vió en secciones anteriores, una web se compone de controladores que mantienen una serie de manejadores. Cada manejador debe indicar el nombre de la vista concreta que debe utilizarse como respuesta a la petición recibida. Spring permite especificar la vista retornando una simple cadena de texto con el nombre de la vista, un objeto de la clase View o ModelAndView, o dejando que se utilice un nombre por defecto construido por convención en vez de por configuración.

En el Script 22 se tiene un ejemplo que muestra un controlador que retorna el nombre virtual de vista "saludo".

Script 22

```
  @RequestMapping(value="/bienvenida")
  public String handler() {
     return "saludo";
  }
```

Para resolver los nombres Spring utiliza un bean que implementa la interface ViewResolver. Algunas de las implementaciones disponibles son por ejemplo XmlViewResolver, que resuelve los nombres usando un fichero XML, o UrlBasedViewResolver, que los resuelve de una forma directa sin necesidad de configuración.

El Script 23 muestra un ejemplo de la configuración de un bean que resuelve los nombres de las vistas anteponiéndoles `"/WEB-INF/jsp"` y terminándolos con `".jsp"`.

Script 23

```
  <bean class="org.springframework.web.servlet.view.UrlBasedViewResolver">
    <property name="viewClass"
value="org.springframework.web.servlet.view.JstlView"/>
    <property name="prefix" value="/WEB-INF/jsp/"/>
    <property name="suffix" value=".jsp"/>
  </bean>
```

De esta forma, la vista `"saludo"` se resuelve a `"/WEB-INF/jsp/saludo.jsp"`. Es conveniente que las paginas JSP estén dentro del directorio "WEB-INF" por seguridad, ya que de esa forma no son accesibles desde fuera del servidor directamente.

Spring también permite configurar varios beans para resolver los nombres de las vistas, formando una cadena. Útil cuando se necesitan distintas tecnologías en una misma aplicación, e incluso resolver hacia vistas distintas en función del tipo de contenido solicitado (HTML, atom, PDF, etc).

6.2 Errores y excepciones

El hecho de utilizar Spring no implica que se tenga que renunciar a la configuración habitual en el fichero web.xml para la gestión de errores o excepciones, tal como se ilustra en el Script 24.

Script 24

```
<error-page>
   <error-code>404</error-code>
   <location>/noEncontrado</location>
</error-page>

<error-page>
   <exception-type>java.lang.Exception</exception-type>
   <location>/excepcion</location>
</error-page>
```

No obstante, Spring permite especificar la anotación `@ExceptionHandler` sobre un controlador para responder a las excepciones de un determinado tipo, tal como se ilustra en el Script 25.

Script 25

```
@ExceptionHandler(IOException.class)
public String handler(IOException ex, HttpServletRequest request) {
   . . .
}
```

6.3 Redirección

En el desarrollo de una aplicación normal basada en web a veces es necesario instruir al cliente para realizar una redirección HTTP. Lo que quiere decir que el cliente acaba solicitando una URL distinta a la de partida. El ejemplo más habitual es el del envío de los datos de un formulario con un POST, que fuerza la redirección con un GET a una página de éxito o error según el resultado. De esta forma se evita que el cliente pueda enviar varias veces los datos del formulario refrescando la página, ya que desde el punto de vista del navegador el cliente se encuentra en una página obtenida con un GET.

Spring permite realizar redirecciones a un manejador retornando un objeto de tipo RedirectView o utilizando el prefijo `redirect:` en el nombre de la vista, tal como se ilustra en el Script 26.

Script 26

```
@RequestMapping(value="/bienvenida", method=RequestMethod.POST)
public String handler() {
   return "redirect:/ala/playa";
}
```

De manera similar, Spring permite utilizar el prefijo `forward:` con el comportamiento acostumbrado. Es decir, redirigir de una página a otra, pero esta vez sin intervención del cliente, de forma que la URL aparentemente sea la misma que la originalmente solicitada.

6.4 Tag library

Spring define sus propias librerías de tags de forma similar a las ofrecidas por JSTL. Pero como de costumbre su utilización es opcional, no se exige su uso.

Spring define actualmente dos librerías de tags. La primera es de uso general, para el manejo de error, soporte para themes, internacionalización y binding, principalmente:

```
<%@ taglib prefix="spring" uri="http://www.springframework.org/tags" %>
```

La segunda es más específica para la gestión de formularios:

```
<%@ taglib prefix="form" uri="http://www.springframework.org/tags/form"
%>
```

6.5 spring:message

Este tag proporciona el texto de un mensaje a partir de su código, resuelto en base a la configuración de internacionalización:

```
<spring:message code='saludo'/>
```

6.6 form:form

Este tag contiene a todos los demás tags de formularios, ofreciendo sobre todo facilidades para el binding.

Consideremos por ejemplo la siguiente clase POJO Cliente que se encuentra en el Script 27.

Script 27

```
public class Cliente {
   private String nombre;
   private String apellidos;
   . . .
}
```

Un controlador puede devolver un objeto de este tipo en el modelo, tal como se ilustra en el Script 28.

Script 28

```
public ModelAndView handler() {
   ModelAndView mv = new ModelAndView("formulario");
   Cliente cliente = new Cliente();
   cliente.setNombre("GUSTAVO");
   cliente.setApellidos("CORONEL");
   mv.addObject("cliente", cliente);
   return mv;
}
```

De forma que la página JSP correspondiente puede utilizarlo directamente, tal como se ilustra en el Script 29.

Script 29

```
<form:form modelAttribute="cliente"
          action="doClienteProcess.htm" method="post">
   <div>
      Nombre: <form:input path="nombre"/>
      <br/>
      Apellidos: <form:input path="apellidos"/>
   </div>
   <input type="submit" value="Enviar"/>
</form:form>
```

El atributo `modelAttribute` establece el nombre del objeto dentro del modelo que contiene los datos en proceso. Por defecto su valor es `"command"` por compatibilidad con JSP, y de hecho, este valor también se puede cambiar utilizando `commandName`.

Los atributos de tipo `path` permiten indicar una propiedad dentro del objeto. De forma que `path="nombre"` se resuelve como `cliente.getNombre()`.

Al hacer click al botón de enviar, el controlador correspondiente a esta petición recibe un objeto de la clase Cliente, tal como se ilustra en el Script 30.

Script 30

```
  @RequestMapping(value="/doClienteProcess.htm",
method=RequestMethod.POST)
  public String handler(Cliente cliente) {
     . . .
  }
```

6.7 Controles

Dentro del tag de formulario se pueden incluir los siguientes controles:

- form:checkbox
- form:checkboxes
- form:radiobutton
- form:radiobuttons
- form:password
- form:select
- form:option
- form:options

- form:textarea
- form:hidden
- form:errors

Todos son equivalentes a sus controles homónimos en HTML, pero con las capacidades de binding añadidas a través del atributo `path`. Además, los controles que tienen el nombre en plural admiten un atributo items que facilita renderizar directamente los valores de una colección, tal como se ilustra en el Script 31.

Script 31

```
<form:select path="aficiones" items="${aficiones}" />
```

7 EJERCICIO 3

Desarrollar una aplicación que permita calcular el importe a pagar a un trabajador independiente.

Para calcular el importe bruto se debe tener en cuenta lo siguiente:

- Cantidad de horas por día trabajados
- Los días trabajados
- Pago por hora

Se debe calcular:

- El importe bruto
- El impuesto a la renta
- El importe neto

8 CAPA DE SERVICIOS

8.1 El application context

Spring ofrece dos clases para cargar un fichero XML con el que instanciar el contexto en una aplicación web. La primera es `ContextLoaderListener`, y la segunda `ContextLoaderServlet`. La única diferencia real entre ambas es que la segunda sólo se puede ejecutar en servidores que cumplan con la especificación Servlet 2.4. Aunque en la práctica, con la primera debería ser suficiente para cualquier aplicación.

Estas clases actúan como listeners, y se inicializan justo después de que el servidor cree el contexto para la ejecución de servlets, momento perfecto para instanciar el objeto ApplicationContext de Spring.

Al tratarse de un proyecto web, la configuración se realiza en el fichero `web.xml`, el Script 32 ilustra lo que sería la configuración del listener.

Script 32

```
<listener>
  <listener-class>
    org.springframework.web.context.ContextLoaderListener
  </listener-class>
</listener>
```

Esta clase intenta leer el fichero XML por defecto de /WEB-INF/applicationContext.xml. Y si no lo encuentra eleva una excepción. No obstante, se puede también configurar para que utilice uno o varios ficheros distintos al usado por defecto. El Script 33 ilustra lo que sería el archivo correspondiente al application context.

Script 33

```
<context-param>
  <param-name>contextConfigLocation</param-name>
  <param-value>/WEB-INF/applicationContext.xml</param-value>
</context-param>
```

Es importante tener en cuenta es que la clase ContextLoaderListener no está relacionada con Web MVC, sólo sirve para inicializar el contexto de una forma conveniente, por lo que resulta útil también para trabajar con Spring cuando se usan otros frameworks de presentación distintos de Spring Web MVC.

8.2 Configuración

El Script 34 muestra un ejemplo de lo que sería la configuración de application context.

Se en primer lugar, la etiqueta `context:annotation-config` habilita el uso de anotaciones.

La etiqueta `context:component-scan` permite configurar el paquete donde spring ubicará las clases de servicios.

También se debe incluir la configuración de la capa de persistencia, pero eso se desarrollará en un capitulo posterior.

Script 34

```
<beans xmlns="http://www.springframework.org/schema/beans"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:context="http://www.springframework.org/schema/context"
   xsi:schemaLocation="http://www.springframework.org/schema/beans
      http://www.springframework.org/schema/beans/spring-beans.xsd
      http://www.springframework.org/schema/context
      http://www.springframework.org/schema/context/spring-context.xsd">

   <!-- Habilita el uso de anotaciones -->
   <context:annotation-config />

   <!-- Configura la ubicación de las clases de servicios -->
   <context:component-scan base-package="pe.egcc.demomvc.service" />

</beans>
```

8.3 Ejemplo ilustrativo

Se tiene una clase de servicios llamada `MateService` tal como se ilustra en el Script 35.

Script 35

```
package pe.egcc.demomvc.service;

import org.springframework.stereotype.Service;

@Service
public class MateService {

   public int calcPromedio(int n1, int n2, int n3){
      int pr;
      pr = (n1 + n2 + n3) / 3;
      return pr;
   }

}
```

Se tiene una clase controladora de nombre `MateController`, que debe responder a la petición de calculaPromedio.htm, tal como se ilustra en el Script 36.

Script 36

```
package pe.egcc.demomvc.controller;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.servlet.ModelAndView;

import pe.egcc.demomvc.service.MateService;

@Controller
public class MateController {

   @Autowired
   private MateService mateService;

   @RequestMapping(value="/calculaPromedio.htm")
   public ModelAndView calculaPromedio(
        @RequestParam("n1") int n1,
        @RequestParam("n2") int n2,
        @RequestParam("n3") int n3 ){
      // Proceso
      int pr;
      pr = mateService.calcPromedio(n1, n2, n3);
      // Preparando el modelo
      ModelAndView model = new ModelAndView("promedioRpta");
      model.addObject("nota1", n1);
      model.addObject("nota2", n2);
      model.addObject("nota3", n3);
      model.addObject("promedio", pr);
      // Retornando el modelo
      return model;
   }

}
```

Como se puede observar, se le esta inyectando la clase de servicio `MateService` para poder hacer el respectivo calculo del promedio.

La vista que mostrará el resultado es la pagina JSP `promedioRpta.jsp`, el Script 37 muestra un ejemplo de lo que sería la programación de esta vista.

Script 37

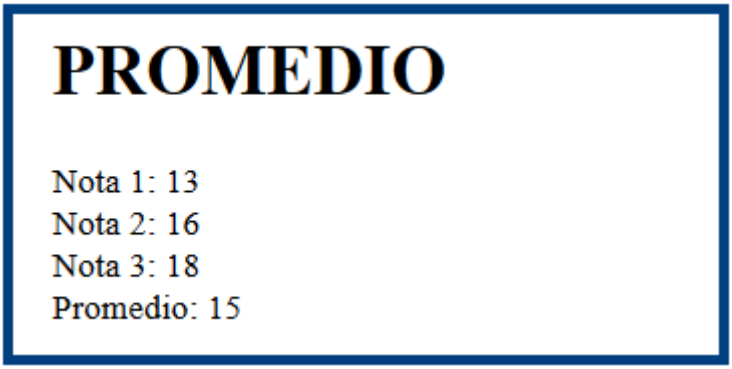
```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
    pageEncoding="ISO-8859-1"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
  <meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
  <title>PROMEDIO</title>
</head>
<body>
  <h1>PROMEDIO</h1>
  Nota 1: ${nota1}<br/>
  Nota 2: ${nota2}<br/>
  Nota 3: ${nota3}<br/>
  Promedio: ${promedio}
</body>
</html>
```

Una URL para hacer esta petición sería la siguiente:

```
http://localhost:8080/appDemo/calculaPromedio.htm?n1=13&n2=16&n3=18
```

También podría hacerse desde un formulario.

Y su resultado sería el siguiente:



9 EJERCICIO 4

Desarrollar un proyecto utilizando Spring MVC que permita calcular el promedio de un estudiante de un detarminado curso, las notas a procesar son:

Nota	Peso
PP: Promedio de practicas	40%
EP: Examen parcial	30%
EF: Examen final	30%

10 OTROS ASPECTOS

10.1 @RequestBody

Esta anotación permite acceder al cuerpo de una petición HTTP, tal como se ilustra en el Script 38.

Script 38

```
@RequestMapping(value="/grabar")
public String handler(@RequestBody Persona bean) {
   . . .
}
```

Se puede hacer que el cliente envíe al servidor un objeto serializado en JSON o XML. Este objeto se envía entonces en el cuerpo de la petición HTTP del cliente y el trabajo de Spring es deserializarlo y "transformarlo" a objeto Java. En el método del controller que responda a la petición simplemente anotamos el argumento que queremos "vincular" al objeto con `@RequestBody`.

Para el caso del Script 38, el argumento `bean` recibirá un objeto serializado de tipo `Persona`.

10.2 @ResponseBody

Esta anotación se utiliza cuando la respuesta será envía directamente al cliente.

Cuando el valor de retorno es un `String`, simplemente se envía el texto correspondiente en la respuesta HTTP. Si es un objeto Java cualquiera se serializará automáticamente. El Script 39 presenta lo que podría ser el esquema de lo que podría ser su implementación.

Script 39

```
  @RequestMapping("/consultarDatos")
  public @ResponseBody String consultarDatos(@RequestParam("codigo")
String codigo) {
     . . .
  }
```

Se aplica, por ejemplo, cuando se necesita retornas datos en formato JSON.

10.3 @ModelAttribute

Esta anotación, cuando se usa a nivel de método indica que el objeto devuelto será establecido como Modelo dentro del ciclo de vida del controlador. Es útil cuando requerimos llenar con datos por defecto la vista. Además permite especificar el nombre del objeto, tal como se ilustra en el Script 40.

Script 40

```
  @ModelAttribute("userForm")
  public User formBackingObject() {
     . . .
  }
```

El nombre del método es de libre elección.

@ModelAttribute puede ser utilizada a nivel de argumento de método, para especificar que al argumento anotado se debe pasar el objeto creado en el método de inicialización, el Script 41 muestra un caso.

Script 41

```
@Controller("userController")
@RequestMapping("/user.htm")
public class UserController {

   @ModelAttribute("userForm")
   public User formBackingObject() {
      . . .
   }

   @RequestMapping(method = RequestMethod.GET)
   public ModelAndView detail() {
      . . .
   }

   @RequestMapping(method = RequestMethod.POST)
   public ModelAndView changePassword(@ModelAttribute("userForm") User
user) {
      . . .
   }

   . . .

}
```

Con esta estructura ya tenemos un controlador básico que atiende peticiones de una URL específica, ya sea por GET o POST e inicializa el objeto modelo.

10.4 HANDLERS

Los handlers (manejadores) son los métodos dentro de los controladores que procesan las peticiones que recibe el servidor web. Spring no estimula que cumplan con el contrato de una interface predeterminada, como ocurre con los servlets tradicionales, ofreciendo un grado de libertad bastante grande a la hora de realizar la implementación.

10.4.1 Tipos de Argumentos

Un handler puede tener cualquier número y casi cualquier tipo de argumentos sin restricciones en el orden, excepto para los de un tipo llamado `BindingResult`.

Los tipos de argumentos soportados por un handler son:

- ServletRequest, HttpServletRequest, ServletResponse, HttpServletResponseInputStream, OutputStream, Reader y Writer, como en los servlets tradicionales.
- WebRequest o NativeWebRequest, que permiten acceder a los objetos de petición y respuesta originales sin tener que utilizar las clases de los servlets tradicionales.
- HttpSession, con la sesión actual.
- Principal, con el usuario actual autentificado.
- Locale, con la configuración regional.
- Parámetros anotados con @PathVariable, @RequestParam, @RequestHeader, @RequestBody o @RequestPart, para acceder a los distintos componentes de una petición HTTP.
- HttpEntity<?>, para acceder a los componentes de un petición HTTP sin usar anotaciones.
- Map, Model o ModelMap, para acceder a los datos de negocio intercambiados con la capa de presentación.
- RedirectAttributes, para especificar los atributos a utilizar cuando se produce una redirección.
- Cualquier tipo de objeto sobre el que se deba realizar el binding entre la información del usuario y el modelo de negocio de la aplicación.
- Errors o BindingResult, con el resultado del proceso de binding efectuado sobre el parámetro declarado justo anteriormente. Es decir, el orden es importante.
- SessionStatus, para controlar el estado e inicializar variables de sesión.

- UriComponentsBuilder, de utilidad para construir URLs relativas a la petición que está siendo atendida.

10.4.2 Tipos Retornados

Un handler puede retornar cualquiera de los siguientes tipos:

- ModelAndView, con la información de proceso y la vista, o nombre de la vista, a aplicar.
- Model o Map, con la información de proceso, siendo el nombre de la vista resuelto con algún tipo de RequestToViewNameTranslator, cuya implementación por defecto lo resuelve eliminando la extensión del fichero de la URL (admin/index.html => admin/index).
- void, si el handler gestiona la propia respuesta a través de un argumento de tipo HttpServletResponse, o el nombre de la vista se resuelve con algún tipo de RequestToViewNameTranslator.
- View o String, con la vista a aplicar, estando la información de proceso determinada por los valores asignados a los argumentos del handler.
- HttpEntity<?> o ResponseEntity<?>, con algún componente HTTP.
- Cualquier tipo, que se interpreta en función de si el handler tiene aplicada la anotación @ResponseBody, en cuyo caso se interpreta como el cuerpo HTTP de la respuesta, o la anotación @ModelAttribute, en cuyo caso se interpreta como un dato de la información de proceso con el nombre indicado en la anotación.

10.4.3 Ejemplos

En el Script 42 se tienen algunos ejemplos de handlers que muestran algunas de las combinaciones permitidas.

Script 42

```
@RequestMapping(value="/tradicional")
public void handler(HttpServletRequest request,
   HttpServletResponse response) {
   . . .
}

@RequestMapping(value="/modelo")
public String handler(Model model) {
   . . .
}

@RequestMapping(value="/binding")
public void handler(@ModelAttribute("receta") Receta receta,
   BindingResult result) {
   . . .
}

@RequestMapping(value="/idioma")
public void handler(Locale locale, @RequestParam("gps") String gps) {
   . . .
}

@RequestMapping(value="/cuerpo")
@ResponseBody
public String handler(@RequestHeader("Accept-Language") String header) {
   . . .
}
```

10.5 MODELOS

Un modelo es un objeto genérico de Spring utilizado para almacenar los datos en proceso durante una petición web. En la práctica es un objeto Map que almacena los valores asociados a una cadena de texto con su nombre.

10.5.1 Model

La forma más sencilla de acceder al modelo dentro de un handler es declarar un argumento de tipo `Model`, `ModelMap` o `Map`, tal como se ilustra en el Script 43.

Script 43

```
@RequestMapping(value="/modelo")
public String handler(Model model) {
   . . .
   model.addAttribute("pi", 3.14159);
   model.addAttribute(calculadora);
   . . .
}
```

Como se observa, los atributos pueden añadirse indicando su nombre, o directamente, sin especificarlo. En este segundo caso Spring asigna nombres automáticamente según los siguientes criterios:

- Un valor nulo elevará una excepción
- Un objeto de tipo com.empresa.Receta tendrá como nombre "receta"
- Un objeto de tipo HashMap tendrá como nombre "hashmap"
- Un array Receta[] con cero o más elementos tendrá como nombre "recetaList"
- Un ArrayList vacío no se añadirá al modelo, pero un ArrayList de recetas con uno o más elementos tendrá como nombre "recetaList"

10.5.2 @SessionAttributes

Esta anotación sirve para acceder o establecer las variables de sesión. Las variables se declaran en la clase, y se accede a ellas a través del modelo, ver Script 44.

Script 44

```
@Controller
@RequestMapping("/recetas")
@SessionAttributes("clienteId")
public class RecetasController {

   @RequestMapping(value="/cuenta")
   public void handler(Model model) {
      model.addAttribute("clienteId", 666);
      . . .
   }

}
```

10.5.3 @ModelAttribute

Cuando se utiliza esta anotación en un método sirve para indicar que el resultado es un atributo que debe añadirse al modelo, tal como se ilustra en elr Script 45.

Script 45

```
   @RequestMapping(value="/consulta/receta/{recetaId}")
   @ModelAttribute("receta")
   public Receta handler(@PathVariable("recetaId") Long recetaId) {
      return recetarioService.getReceta(recetaId);
   }
```

Cuando se utiliza en un argumento de un método sirve para indicar que el atributo debe recuperarse del modelo, tal como se ilustra en el Script 46.

Script 46

```
   @RequestMapping(value="/elabora/receta/{recetaId}", method =
RequestMethod.POST)
   public void handler(@ModelAttribute("receta") Receta receta,
BindingResult result) {
      . . .
   }
```

El valor que se obtiene puede provenir de varias fuentes. En primer lugar puede venir de un valor almacenado en las variables de sesión. En segundo lugar puede venir de un valor que ya se encuentre presente en el modelo, añadido por otro handler del mismo controlador. En tercer lugar puede venir recuperado a partir de un valor presente en la URL, explicado a continuación. Y por último, puede venir instanciado por su constructor por defecto.

Los valores recuperados a partir de un valor presente en la URL se basan en el uso de las facilidades de conversión de tipos de Spring. En el ejemplo, si llega un identificador de receta en forma de `String`, y se define un conversor del tipo `String` al tipo `Receta`, entonces se instanciará un objeto automáticamente a través del conversor. Los tipos básicos por su parte son automáticamente convertidos.

10.5.4 @InitBinder

Esta anotación sirve para declarar un conversor de tipos dentro de un controlador. Se aplica a un método que admite un argumento de tipo `WebDataBinder`, además de casi cualquier combinación de los mismos tipos soportados por los métodos marcados con `@RequestMapping`. La clase `WebDataBinder` permite configurar conversores a medida de una aplicación.